MIRAGE+ A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers

Brett D. Fleisch[*] Randall L. Hyde[†] Niels Christian Juul[‡]

DIKU-rapport 94/13[§]

April 1994

D I K U
Department of Computer Science
University of Copenhagen
Denmark

[*]Department of Computer Science, University of California, Riverside (brett@cs.ucr.edu). This research is sponsored by a Joint Study with IBM Corporation, NSF CDA09209405.

[†]Department of Computer Science, University of California, Riverside (rhyde@cs.ucr.edu).

[‡]Department of Computer Science, University of Copenhagen, Denmark (ncjuul@diku.dk). Work done during PostDoc employment with UCR, also supported by the Danish Natural Science Research Council.

[§]Released simultaneously as UCR-CS-94-2 by University of California, Riverside.

Abstract

We describe the evolution of a *distributed shared memory* (DSM) system, Mirage, and the difficulties encountered when moving the system from a UNIX-based[1] kernel on the VAX to a UNIX-based kernel on personal computers. Mirage provides a network transparent form of shared memory for a loosely coupled environment. The system hides network boundaries for processes that are accessing shared memory and is upward compatible with the UNIX System V Interface Definition.

This paper addresses the architectural dependencies in the design of the system and evaluates performance of the implementation. The new version, MIRAGE+, performs well compared to Mirage even though eight times the amount of data is sent on each page fault because of the larger page size used in the implementation. We show that performance of systems with a large page size to network packet size can be dramatically improved on conventional hardware by applying three well-known techniques: packet blasting, compression, and running at interrupt level.

The measured time for a page fault in MIRAGE+ has been reduced 37% by sending a page using packet blasting instead of using a handshake for each portion of the page. When compression was added to MIRAGE+, the time to fault a page across the network was further improved by 47% when the page was compressed into one network packet. Our measured performance compares favorably with the amount of time it takes to fault a page from disk. Lastly, running at interrupt level may improve performance 16% when faulting pages without compression.

[1]UNIX is a Registered Trademark of Unix International.

Contents

List of Tables

List of Figures

1 Introduction

Loosely coupled distributed systems have relatively low bandwidth inter-site communication when compared to tightly coupled processors or uniprocessors. Achieving good performance presents a number of challenges to designers, particularly when coordinated sharing is desired. Nevertheless, loosely coupled distributed systems provide the potential to scale, economically, to very large configurations using commodity hardware.

In the past, operating system designers have exploited the similarity between network packets and messages in the design of loosely coupled distributed systems[Accetta 86]. However, some researchers have observed that the message passing approach may not be well suited for tightly coupled processors that access shared memory[Li 86b, Li 90, Bisiani 90, Bennett 90b, Ramachandran 88, Fleisch 89b]. Message passing interfaces require the programmer to use conceptually different primitives and organize their code differently than shared memory interfaces. Further, communicating large, complex data structures may be difficult or inefficient using message passing. An alternate approach is to use *Distributed Shared Memory* (DSM)[Fleisch 89b, Nitzberg 91].

Since large address space machines are becoming ubiquitous, globally-accessible memory is increasingly important. DSM systems permit sites to access common blocks of memory using the convenience of load and store instructions. Nonetheless, DSM presents challenges to the system designer since DSM requires well tuned components to support efficient transfer between sites.

In this paper, we focus on performance issues related to DSM page size and the underlying DSM support structure. This work is based on our previous DSM system called Mirage[Fleisch 89b, Fleisch 89a]. Mirage is a DSM facility implemented entirely in the kernel of a UNIX-based operating system[Popek 81, Walker 83]. Here we examine our work to adapt Mirage for personal computers with a larger page size. We call the new system MIRAGE+.

1.1 Goals and Overview

Our prior work[Fleisch 89b] focused on operating system extensions to support DSM, performance of synthetic applications which exercise DSM, examination of supporting algorithms and protocols, and performance optimizations using a *time-based locking* approach.[2] Although experiences with Mirage were encouraging, there were a number of concerns that motivated our port to a new research platform: 1) the hardware running the Mirage prototype (VAX 11/750s) was obsolete, 2) Mirage was built on an early version of the Locus operating system[Popek 81, Walker 83] that operated only on VAXs, 3) we needed a testbed where we could examine the scalability of our system beyond three machines, and 4) we needed a new platform to address the issue of reliability in our future work.

This paper addresses our experiences moving DSM to a new platform. The paper reports on factors that affect the design, implementation, and component performance in MIRAGE+. Our goal was to design a high performance DSM system. However, an important design constraint is that our system be portable. Most of the MIRAGE+ code is machine and host operating system independent assuming a kernel-to-kernel packet transport mechanism is available. The few machine dependent portions of the code are isolated in separate modules.

The larger page size supported by the hardware was the source of serious performance problems. Portability can be severely limited if a large page size impacts performance. Our goal is to be able to add DSM to a system as a drop-in component without spending considerable time optimizing vendor supplied communication subsystems that may not handle

[2]Formerly called time-based coherence.

large page sizes well. We address performance problems using various techniques including: high level packet blasting, shipping less data via compression, and running at interrupt level.

The larger page size also introduces the possibility of a larger impact from *false sharing*. In this paper, our major concern is to measure the DSM cost per page fault. We address performance in terms of number of control messages to obtain data in the MIRAGE+ protocol, i.e., time to obtain remote pages and move the pages to service the requests. So, we focus on component costs and the operation and performance of the protocol rather than reduction of the number of page faults. Our performance enhancements are independent of the amount of false sharing from the application level. The enhancements improve the performance of all applications including those that exhibit false sharing.

1.2 The New Environment

The environment consists of 12 PS/2 Model 70s and 80s upgraded with i486 CPUs with 10 to 16 Mbytes of main memory.[3] The cluster has a total disk storage capacity just under 5 Gbyte. The systems are connected by a 10 Mbps Ethernet[Metcalfe 76] using Ungermann-Bass NICps/2 network adapters (technology circa 1987) without on-board caching. They execute our modified AIX Version 1.2 which includes IBM's Transparent Computing Facility (TCF).[4]

As part of the new work we have completed, we have instrumented the operating system and the DSM system with optional timers and counters to measure performance. These mechanisms are discussed further in the beginning of the performance section. Second, we have improved the performance of the basic network page faulting in MIRAGE+. Third, we have added page compression. These last two mechanisms present two additional steps into our research into high performance DSM suitable for other high performance architectures.

2 The Mirage Model of Distributed Shared Memory

The new DSM system, like Mirage[Fleisch 89b], uses a paged segmentation scheme[Daley 68, Denning 70]. In our model, processes create shared memory *segments* by specifying the size of the segment, name, and access protection. Processes locate and *attach* segments into their virtual memory address space by name. Once the named segment is attached, the shared memory behaves as conventional memory, the only difference being that changes to the underlying memory are visible to the other local or remote processes sharing the segment.

A comparison of the Mirage model to other DSM models is given in the related work section at the end of this paper.

2.1 Consistency Control and Coherence

Consistency control is an important issue in our model of shared memory. At the outset of the design we decided that it would be unacceptable for processes to read data that has become out-of-date or *stale*. In our model, we provide *coherence* at the lowest system level. A coherent implementation is one in which a store to an address is always visible to all subsequent load operations of the same address, independent of the machine location where the load occurs. Higher level synchronization primitives, such as semaphores or monitors[Brinch Hansen 73, Dijkstra 72, Hoare 74] may be used by applications that require

[3]The maximum number of uniformly configured machines in the cluster was eight.

[4]AIX/TCF is a trademark of IBM. The TCF portion of the AIX product has since been discontinued.

additional consistency guarantees. For example, we use the UNIX System V semaphore interface in many of our (distributed) applications.

Our past work on Mirage focused on a *time-based locking approach* to DSM. The system uses a clock mechanism to control when a site may be interrupted from its read/write processing to relinquish shared pages. The clock mechanism grants the readers or the current writer a *time window* (Δ) during which the processes on a given site possess the page. Like the traditional CPU time slice, Δ is used to apportion time for the page to the site(s). During the time window, processes on the site(s) having read-only access may read, or processes at the site having write access may read or write the page. The page may also be unused during Δ.

The time window provides a degree of fairness between sites requesting page access, the current site using the page, and the controlling site (library) which attempts to invalidate the page on behalf of another requester. In a sense, Δ provides some degree of control over the *processor locality*, the number of references to a given page a processor will make before another processor is allowed to reference that page.

2.2 The Mirage+ Protocol

When processes are co-located and they share memory, the UNIX System V implementation is used. However, when remote memory access is required, the MIRAGE+ protocol is used. In MIRAGE+ a *load* is implemented by reading a local copy of the data page, if present. However, since MIRAGE+ is a write-invalidate coherent system, a *store* requires that all read-only copies of a page be invalidated before storing to the page. In this case, a writable copy of the page is required in the network and all other readable copies of the page are discarded.

In MIRAGE+ there is one distinguished site associated with each segment, called the *library site*. Requests for pages are sent to the library site, queued, and sequentially processed. All pages must be "checked out" from the library. The library instructs the site which possesses the page to return it directly to the requester. Depending on the configuration, there may be several different sites used as library sites for the various segments created by user programs. In MIRAGE+, the site that creates the segment is configured as the library site for the segment.

Another distinguished site in our model is the *clock site*. The clock site is the site that has the most recent copy of a page. For example, if there is a writer for the page on the network, it is chosen as the clock site. On the other hand, if there are a set of readers using the page simultaneously, one of the readers is selected as the clock site. The clock site controls the decision-making for the time-based locking mechanisms described earlier and stores whether the cluster possesses the page in read-only or read-write mode. The library site records which site is acting as clock site.

As mentioned earlier, there may only be one writable copy of a given page in the network at any one time. While there may be multiple readers simultaneously using a page, there may not be read copies at the same time as the write copy. The process of converting a reader to a writer when a protection fault occurs is called an *upgrade*. The process of converting a writer to be a reader is called a *downgrade*. In MIRAGE+, a downgrade occurs when a writer possesses the page and a remote read fault is serviced.

3 The Impact of a Larger Page Size

Perhaps the most significant issue in the transition from Mirage to MIRAGE+ was the larger page size. Mirage had a page size of 512 bytes while MIRAGE+ has a page size of 4096 bytes. This factor of eight difference in page size has had a significant impact on performance. To assemble one MIRAGE+ page requires receipt of four network packets and processing by an interrupt level server process or an interrupt service routine. The AIX/TCF communication subsystem limits each network packet to contain a data buffer of at most 1 Kbyte. The data buffer is in addition to the standard AIX/TCF header information of approximately $\frac{1}{4}$ Kbyte. The limit is imposed by the upper layers of the kernel network software; the lower layer communication software for the Ethernet is limited to roughly 1500 bytes.

Our first implementation of MIRAGE+ addressed the 4 Kbyte page size versus the 1 Kbyte network packet size by modifying the sending routine to split the page into four network packets. The receiving routine gathered the four packets in a buffer and re-assembled the page. Each of the packets was sent using a synchronous *netsend()* routine that required a high-level acknowledgment for each packet.

We measured the performance of the system, and isolated the cost of a remote page fault, which included the transfer of a page between sites. This measured cost averaged 31.7 msec.

3.1 Performance Improvement Techniques

We considered several possible ways to improve performance. Here we discuss the most promising of the optimizations feasible for our environment. These optimizations include:

1. packet blasting
2. running at interrupt level
3. compression

3.2 Technique 1: High-Level Packet Blasting

One possible performance improvement is to optimize the network code to reduce time spent moving data through the software layers of the protocol. The AIX/TCF kernel supports cluster communication that provides virtual circuits between cluster sites. The low-level communication subsystem is responsible for transmitting the programmer's packet to the destination site. The messages are sequenced, ordered, reliable, and possibly combined with other messages when sent to the destination.

Vertical optimization of the protocol layers is a well proven technique to achieve better performance[Schroeder 90, Tanenbaum 92]. Our design constraints were, however, to build the DSM system on top of the communication layers without spending considerable time optimizing communication subsystems written by others.

With high-level packet blasting[Zwaenepoel 85, Carter 89], we expected the total time for a remote page fault would improve significantly. The savings using high-level packet blasting instead of explicitly handshaking each packet, is not only due to the removal of explicit acknowledgments, but also due to increased parallelism during communication. We have applied packet blasting to the protocol for page transmission as well as to our iterative-send version of multicast.

3.3 Technique 2: Interrupt Level Execution

Even with high level packet blasting, communication costs could remain high. Another enhancement was inspired by the fact that processing on the interrupt-level could potentially improve performance. Rather than scheduling and running a kernel-internal server process to process each incoming network message, we process each packet directly in the interrupt service routine. This eliminates the small start-up latency and the potential interference between server processes scheduled concurrently. However, this approach requires additional complexity in providing consistency of the kernel state during the interrupt service.

3.4 Technique 3: Compression

Compression is another technique we use to improve performance. Compression works by reducing the number of network packets that the system must transmit on each page fault. However, compression used inappropriately may actually reduce performance. This occurs when it takes longer to compress, decompress, and transmit the data than to transmit the uncompressed data. Furthermore, some pages cannot be compressed and therefore increase the cost of transmission. Nevertheless, we felt a high performance compression system would improve the expected performance for many applications in MIRAGE+.

4 Performance Analysis of Component Costs

In this section we describe our methodologies, analysis, and results of MIRAGE+ performance costs. We use the time spent servicing a remote page fault as the basis for understanding the overall performance in the system. We compare the costs in Mirage to MIRAGE+. For MIRAGE+ we examine the costs with and without packet blasting, and with and without interrupt level optimization.

4.1 Mechanisms for Performance Measurements

We have instrumented the kernel with a micro-timing function that makes it possible to read a timestamp with the granularity of 1 μsec. Timestamps are buffered inside the kernel and can be output when the kernel is running. Each timestamp call requires approximately 15 μsecs.

Counters have been added to critical routines in the kernel to count function calls. Both the counters and the timestamps may be returned with a system call. In addition, we have instrumented our applications to use kernel timestamping. Instead of printing the actual values of the counters when the kernel is executing, which would interfere with the actual timings themselves, the values may be read at the beginning of the application and again at the end. This enhancement permits the incremental values during the application run to be accumulated and output later.

4.2 Basic Communication Costs

We measured the basic cost of communication between two sites by instrumenting the system call *probe()*. The probe call creates a message in the kernel that is sent though the protocol layers of the communication system at the sending site, over the wire, through the protocol layers at the receiving site from where it is returned to the caller through these same layers. The probe takes approximately 5.5 msec round-trip. Although a probe mimics a high-level ping operation, the probe is somewhat different in that it travels through all of the layers in the communication system, whereas a ping may not.

Based on the timings above, we determined the cost of a short message sent one-way through the communication subsystem to be 2.7 msec. We also measured the round-trip cost of a short message with a long message response to be 6.7 msec. Thus, the cost of a long message sent one-way is 4.0 msec. This includes the additional cost of transmitting 1 Kbyte over the wire (0.8 msec using full bandwidth of a 10 Mbps Ethernet) and copying of the data through the protocol layers of the communication subsystem (0.5 msec).

1. Transfer page request to library site:
   (a) Send request to library site
   (b) Communication of short message
   (c) Receive request at library site
2. Process request at library site
3. Transfer page to user site in four packets:
   (a) Send packet
   (b) Communication of long message
   (c) Receive packet
   (d) Send acknowledgment
   (e) Communication of short message
   (f) Receive acknowledgment
4. Install page at user site and resume the faulting process

Figure 1: Remote Page Fault Algorithm (sketched)

The component costs of a remote page fault for a checked-in page[5] include time at the faulting site, time at the library site, and network communication cost. The last component, includes the cost of sending a message through the communication subsystem at the sending site, over the wire, and through the communication subsystem at the receiving site.

Our detailed measurements show that a page fault request can be sent from the faulting site to the library site in 3.0 msec. This is equivalent to the short message one-way cost of 2.7 msec plus 0.3 msec overhead to trap the page fault and generate the network message. The measurements of the cost to transfer the page in four network packets include the cost of a long message one-way (4.0 msec) and the cost of generating the network message and copying 1 Kbyte of data at each site (0.3 msec to send and 0.2 msec to receive) for each packet. When using high-level acknowledgments for each packet, a cost similar to a short message one-way is added (2.7 msec communication cost) for each acknowledgement. Thus, each part of the page costs 4.5 msec without, and 7.2 msec with, the high-level acknowledgment.

4.3 The Cost of a Page Fault

A remote page fault results in a sequence of actions. The sequence which constitutes the algorithm for a remote page fault for a checked-in page is outlined in Figure 1. Of these action, only those in the critical path contribute to the latency introduced when a process traps due to a remote page fault. The measured latency indicates that the components in the critical path are the sole contributors to the cost. Thus, the cost of sending the last acknowledgment (Steps 3d, 3e, and 3f in Figure 1) adds only the initial transmission time at the user site. We measured that cost to be 0.9 msec.

The measured time for a remote page fault is 31.7 msec as shown in Table 1. This cost is for a site that requests a page checked-in at the library site. The major cost for faulting a page over the network is the cost of sending the page as four packets. The exchange of three of the four packets using the handshake protocol costs 7.2 msec per packet.

[5]Checked-in means that library site is also the clock site for that page.

Operation	User Site (msec)	Communication (msec)	Library Site (msec)	Totals (msec)
1. Transfer Request	0.3[1]	2.7	0.0	3.0
2. Process request			1.4[1]	1.4
3.1 Transfer page (part 1)	0.2[1]	6.7	0.3	7.2
3.2 Transfer page (part 2)	0.2[1]	6.7	0.3	7.2
3.3 Transfer page (part 3)	0.2[1]	6.7	0.3	7.2
3.4 Transfer page (part 4)	0.2[1]	4.7	0.3	5.2
4. Installing page	0.5[1]			0.5
Total	1.6[1]	27.5	2.6	31.7[1]

Table 1: The Component Costs of a Page Fault in MIRAGE+(handshake)

1. indicates a directly measured cost, the remainder is determined analytically.

Operation	User Site (msec)	Communication (msec)	Library Site (msec)	Totals (msec)
1. Transfer Request	0.3[1]	2.7	0.0[1]	3.0
2. Process request			1.4[1]	1.4
3.1 Transfer page (part 1)	0.2[1]	4.0	0.6[1]	4.3
3.2 Transfer page (part 2)	0.2[1]	4.0-1.3=2.7[2]	0.3	3.3
3.3 Transfer page (part 3)	0.2[1]	4.0-1.3=2.7[2]	0.3	3.3
3.4 Transfer page (part 4)	0.2[1]	4.9-1.3=3.6[2]	0.3	4.1
4. Installing page	0.5[1]			0.5
Total	1.6[1]	15.7	2.6	19.9[1]

Table 2: The Component Costs of a Page Fault in MIRAGE+(blast)

1. indicates a directly measured cost, the remainder is determined analytically. 2. Approximately 1.3 msec of the sending is overlapped with the transmission and reception of the previous packet due to packet blasting.

4.4 Packet Blasting Performance

With high-level packet blasting, the total time for a remote page fault was reduced from 31.7 msec to 19.9 msec as shown in Table 2. The savings using high-level packet blasting instead of handshaking is not only due to the removal of explicit acknowledgments, but also due to the parallelism achieved in sending the last three of the four packets. Recall that the network time is attributed to time spent mostly in the communication layers at the sending and receiving sites. Thus, the measurements reflect that when the first packet has left the library site it can be processed at the user site (both by the communication layers on that site and later, by the actual packet receiving code), while the second packet is being sent from the library site simultaneously. We measured the average time to receive the second and third packet as 3.2 msec, the first packet does not benefit from any concurrency, and the last packet must reply with a high-level acknowledgment for all four packets.

4.5 Comparision of Remote Page Fault Costs

We have compared the results with previous published results for Mirage[Fleisch 89b] running on a set of three VAX 11/750s. The comparison is shown as Table 3. The component costs of

Operation Time (in msec)	Mirage	MIRAGE+ (handshake)	MIRAGE+ (blasting)
1a Read request[1]	2.5	0.3	0.3
1b Request transmission[2]	3.2	1.3	1.3
3 Page Transmission[2]	7.5	13.4	7.5
4 General overhead[1]		0.5	0.5
Total user side	13.2	15.5	9.6
1b Request transmission[2]	3.2	1.3	1.3
1c Request received[1]	1.5	0.0	0.0
2 Process request[1]	2.0	1.4	1.4
3 Page Transmission[2]	7.5	13.4	7.5
Delay by library site	14.2	16.1	10.2
Total costs[1]	27.5	31.7	19.9
Short message, round-trip[1]	12.9	5.5	
Long message(1K buffer) w/short response[1]	21.5	7.2	

Table 3: Comparison of Page Fault Costs between Mirage and MIRAGE+
Mirage was measured on VAX 11/750 with 512 byte page size, and MIRAGE+ was measured on i486/25MHz machines with 4K page size using 4 1K network packets. [1] indicates a directly measured cost, the remainder is determined analytically. [2] indicates a transmission delay including the sending and receiving cost. Half of the cost is attributed to each of the sites.

the MIRAGE+ measurements are shown in a composite form similar to the VAX measurements of Mirage, where the component costs are attributed to either the user or library site. With high-level packet blasting added to the kernel, MIRAGE+ compares favorably given the eight times page size difference. Actually most of the time spent during a remote page fault is attributed to the communication subsystem, as in Mirage.

4.6 Interrupt Level Execution

Even with packet blasting, communication costs remain high. Another enhancement we pursued was inspired by the fact that processing on the interrupt-level may improve performance. This approach is an alternative to scheduling and running a kernel-internal server process. By processing the packet reception activities directly at interrupt-level instead of through a server proc, we reduced the total execution time for a page fault by approximately 4 msec. The additional time consumed at interrupt time is minimal compared to the time already spent passing through the network layers. Consequently, other interrupts were not blocked out for a long period. More importantly, we did not observe any dropped or overrun packets that could have arisen from using this technique. When the last packet is received, however, we must assemble the page, install it, and resume the processes waiting for the page. Thus, because of the additional time required to process the last packet, we chose not to use interrupt level execution for that packet, but only on the first three packets. This optimization reduced the page fault time to 16.7 msec (from 19.9 msec), a performance enhancement of 3.2 msec, or 16%.

4.7 Cost of Invalidation

In the previous sections we described the cost of a page fault without considering the cost of invalidating outstanding readers when a write fault occurs. The invalidation expense would be required for any write-invalidate DSM system.

In Mirage+ the system multicasts an invalidation message to all readers being discarded. The packet blasting technique is used to implement an iterative send version of multicast. Each multicast invalidation message is sent to all of the designated sites and then all the high level responses are obtained. The cost for an invalidation of one, two, three, and four additional sites was measured as 4.6 msec, 6.8 msec, 7.0 msec, and 8.2 msec respectively. These measurements should scale if more sites need to be invalidated; the cost of invalidation should increase linearly in the number of concurrent readers.

With our simple, iterative (and blasting) multicast on top of the existing communication subsystem, the invalidation of multiple readers is as efficient as one can hope for in a system that does not support true multicast. However, our measurements emphasize the importance of having a high performance multicast available in the communication subsystem. In selecting alternate networking technologies, such as ATM technology, support for true multicast is essential for DSM systems.

4.8 Compression

Compression attempts to reduce the number of network packets that the system must transmit on each page fault. In our system, the performance of the network protocol layers reduces system performance significantly. Currently, the system does allow only two sizes of packets, a short message (containing the message system header only), and a long message (with an additional 1 Kbyte data buffer). Thus, the difference in transmission time for various sized data buffers does not pay off, unless it can be reduced to zero. Therefore, reducing the size of transmitted page does not necessarily improve performance unless it eliminates entire packets. Since the number of messages has a key role in page fault performance, a compression scheme which works best with Mirage+ is one which produces compression ratios of less than 75%, less than 50%, or less than 25%.

The time to compress and decompress the data *increases* the network latency. To reduce communication time, compression must save more transmission time than it costs to compress and decompress the data. Clearly a very fast *codec* (compressor/decompressor) which does not compress the data well will not improve performance. Likewise, an algorithm which compresses the data significantly will not improve overall performance if it takes too long. Therefore, it is important to consider *both* speed and expected compression ratio when choosing codec algorithms for distributed memory systems.

There are many codec algorithms available or published in the literature[Bell 89b, Raita 87, Ziv 77, Williams 90, Bell 89a, Storer 88, Nelson 91]. For the purposes of this initial work on whether compression would be useful, we chose to use run-length encoding (RLE) as our codec of choice. RLE exhibits many of the good characteristics required from a Mirage+ codec: it is fast, compression ratios do not depend on block size, and it is very easy to implement. Though RLE does not produce outstanding compression ratios on common data, it works quite well with the benchmarks we are currently using. Its simplicity gave us the opportunity to instrument our system and run several experiments to check the feasibility of using compression within Mirage+.

On our systems running compression, the current RLE codec algorithm averages between 0.9 to 2.0 msec to process a 4K block of data. If the RLE codec manages to compress to at least 75%, it improves the performance of Mirage+.

Packets required to send the page		1	2	3	4
Compression	msec	0.5	0.9	1.0	1.3
Decompression	msec	0.4	0.5	0.7	0.0
Compression overhead	msec	0.9	1.4	1.7	1.3
Costs minus compression overhead	msec	9.6	12.1	15.4	20.0
Run-time using compression	msec	10.5	13.5	17.1	21.3
Speed-up by compression	percent	47.2	32.2	14.1	-7.0

Table 4: Average Cost of Remote Page Fault with Compression

Table 4 shows the average page fault time when using compression on various pages that compress into one, two, three, or four packets. Note that if the compression algorithm cannot achieve at least a 75% compression ratio, it immediately aborts and transmits an uncompressed page.

The combination of our slow Ethernet cards, slow CPUs, and AIX's slow network communication software could provide an overly optimistic picture of the benefits of using compression. An issue is whether compression improves performance if network latency were reduced. Figure 2 shows the effect of compression performance on our system if we reduce network latency. The x-axis is the amount of software and hardware latency for our communication sub-system on a 10 Mbps Ethernet. This cost includes passing the message which includes the buffers to the network layers, through the layers, over the wire, and through the layers on the destination. The current network latency for a remote page fault on our system is 15.7 msec when using packet blasting (See Table 2). This corresponds to the network cost of one short message of 2.7 msec, four long messages at 4 msec each partly overlapping, and part of the sending cost for a short message, 0.9 msec. The current communication latency is 2.7 msec for a short message sent one-way. The zero point on the x-axis corresponds to a bus which is capable of transmitting all the messages without delay. Although not possible in practice, this isolates the MIRAGE+ protocol overhead to be approximately 3.6 msec.[6] The y-axis indicates the total time for a compressed remote page fault, including compression/decompression time. If the codec succeeds in compressing the page to one packet, the system will always perform better. If the codec reduces the transmission to two packets, then the codec improves system performance if the latency in the communication system is at least 0.3 msec per short message sent one-way. For three packets, there must be about 0.9 msec latency per short message sent one-way for the system to show improvement.

As Thekkath and Levy[Thekkath 93] point out, bandwidths are improving dramatically while latencies are not. This suggests that network transmission speed is limited by network components other than wire time such as the network protocol layers. There are two obvious ways to improve the performance of the network protocol stack: recode the network layers or use a faster CPU. Rewriting the network code is beyond the scope of the MIRAGE+ project, although, as the graph indicates, one would have to reduce the latency by a considerable amount to eliminate the benefits of compression. Increasing the speed of the CPU will certainly reduce the the network latency, but it improves codec performance as well.

[6] 1.0+2.6=3.6, taking advantage of concurrency between the sending and the receiving even when no delay is present.

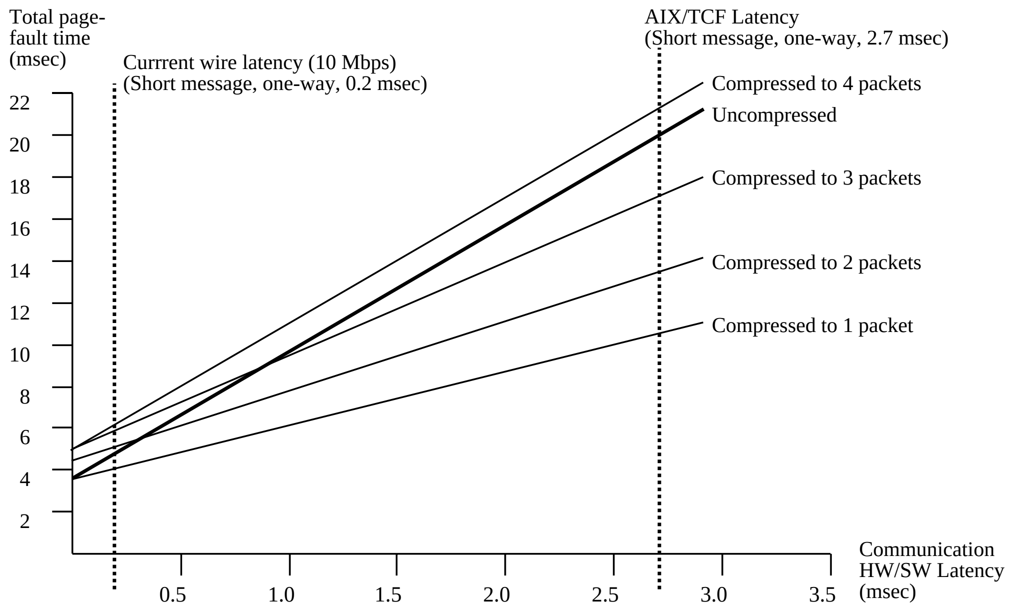

Figure 2: Codec Performance vs. Network Latency
Average cost of remote page fault under varying network latencies.

5 Performance Evaluation of Applications

This section provides an analysis of measurements of intense DSM use by five applications, with and without compression.

5.1 Philosophy Behind the Methodology

To study DSM performance we constrain our attention to issues associated with component costs of the DSM system. We focus on designing a system whose components are efficient. To examine how closely these objectives have been met, we designed and implemented our test software to isolate DSM memory access costs from other non-germane system components such as file access, naming, and I/O device access. Nevertheless, even isolating these aspects, an application's locality, site locality (for DSM), and application programming style can greatly influence performance. To address the problems that could arise in instrumenting non-germane functions in the system, we focus on timings from the direct delivery of DSM services. For example, most of the applications synchronize using the memory system itself.

False sharing can have a big impact on the performance of a DSM system. False sharing phenomena arises from the nature of applications. Dynamic detection and attribution of false sharing remains a hard, important research problem. This issue is beyond the scope of this paper. We present a precise definition of false sharing and our work in the area in a separate paper[Hyde 94].

5.2 Application Descriptions

To check the viability of compression before investing extensive work in it, we constructed three applications which access memory using common access patterns: *EachN*, *UptoN*, and

Step	Action (memory-oriented part of algorithm)	Shared memory access
P1.	Scan through the battle field searching for a new shot by our adversary.	**Read** entire own field multiple times.
P2.	Acknowledge the shot by writing 'hit' or 'miss' at the location of the adversary's shot in the battle field.	**Write** once to one place on own field.
P3.	Shoot into our adversary's battle field.	**Write** once to one place on other field.
P4.	Wait for our adversary to write a 'hit' or 'miss' value at the location of our shot.	Multiple **reads** of one place in other field.

Figure 3: The Memory Access Pattern of the Central Loop in Battleship Simulation

RandN. Each of these applications starts with a page containing all zeros and writes uncompressable values to the page. EachN writes to each n^{th} byte in the page. UptoN writes to the first n bytes of the page. RandN writes n randomly chosen bytes to the page. We ran these applications for different values of n.

RandN simulates arbitrary writes throughout a page. It also helps simulate the effects of false sharing which occurs in many applications. UptoN simulates DSM applications which allocate some amount of memory less than one 4K page. For example, UptoN with N equal to 128 simulates a DSM application which allocates a 128 byte block of shared data. The remaining bytes in the shared page remain unused, the result of internal fragmentation. EachN simulates accesses to the columns of an array when using row major ordering such as zeroing out an array in memory or writing the result of a matrix multiply.

The *Battleship Simulation*, a version of the Milton Bradley board game, is a very useful application to exercise DSM. The battleship simulation features two competitors. Each player acts as an adversary attempting to destroy the other by shooting into a battle field in shared memory. Our implementation has each competitor execute as a separate process. Synchronization between the two processes uses the memory system itself.

The simulation works as follows: each competitor scans the battlefield repeatedly until they locate a shot made by their adversary, which is then acknowledged by writing either 'hit' or 'miss' in the playing field. The competitor then shoots into the battle field and waits for the other party to acknowledge the shot. This process repeats until one player gets a sequence of shots that constitutes a "win". At that time the simulation concludes. Figure 3 shows the algorithm in detail.

Matrix multiply is another application which exercises DSM. Unlike Battleship, which is a worst case application for two sites, the matrix multiply program runs well under DSM. To test the behavior of our system, especially the cost of communication, we ran several experiments on different matrices choosing sizes which produced compute-bound executions and I/O bound executions (See Figure 6).

The matrix multiplication algorithm computes C := A $\times$ B on integer arrays. In our experiments we used between two and seven sites. The first site loads the A and B arrays, instructs the other site(s) to perform the multiplication on their portion of the array, and then collects the results. We divided the work evenly among the remaining sites by having them work on a band of rows (e.g., for a 240×240 matrix with six workers, each site handled a band of 40 rows in the matrix). We used a separate shared memory segment to synchronize the starting process and the workers.

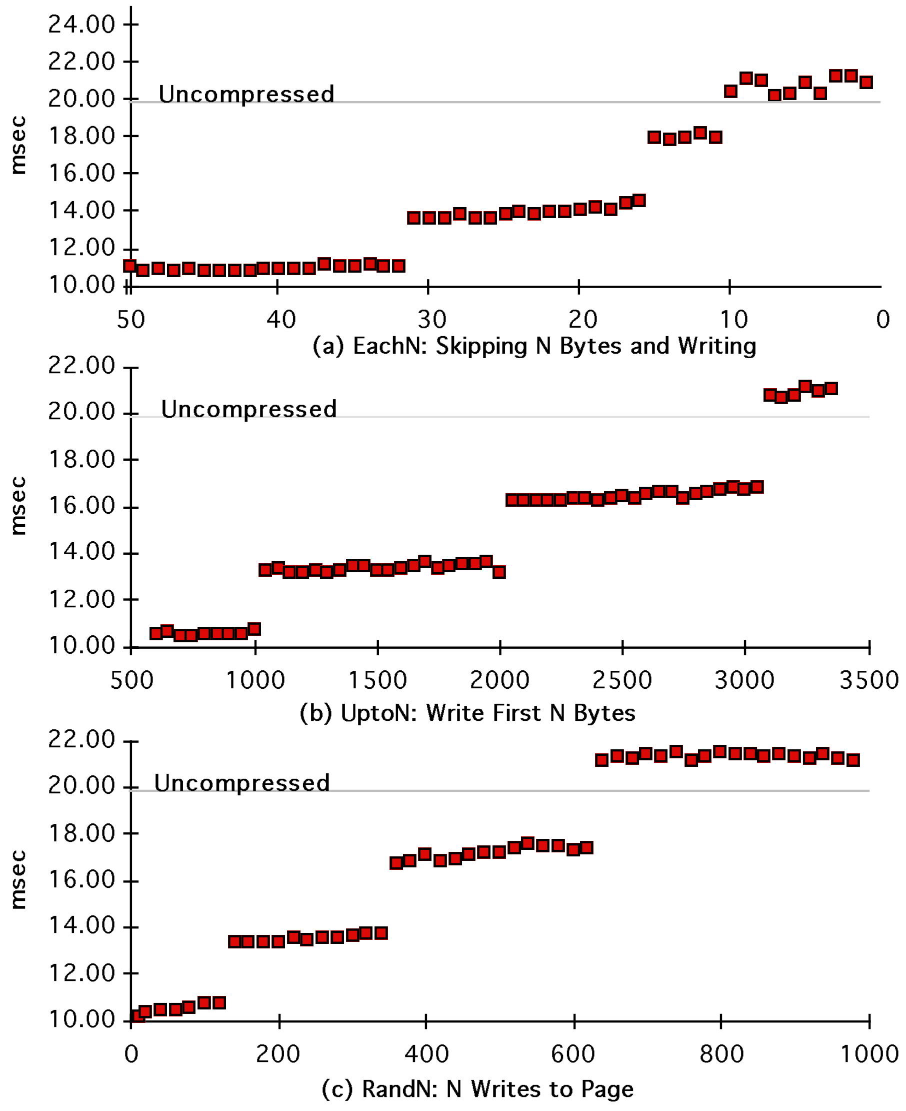

Figure 4: The Effect of Compression on Remote Page Fault Time
(a) scrambling every N'th byte, (b) scrambling the first N bytes, and (c) scrambling N bytes at random.

Extra elements were added for padding at the end of each row to eliminate the effect of false sharing for matrices larger than 120×120. This padding required some additional transmission time, but this was less than the time lost to false sharing.

5.3 Performance of EachN, UptoN, and RandN

To test the performance of MIRAGE+'s codec we ran the EachN, UptoN, and RandN applications. The results appear in Figure 4. If EachN stores an uncompressable byte every eight bytes, it will produce a data block which cannot be compressed. MIRAGE+'s RLE codec needs at least two adjacent double words in order to compress anything. At N=10 the codec performs poorest. At this point the codec has to process a large part of the 4K buffer before determining that it cannot compress the data below the 75% point. At N=11 the compression time is still high, but the codec achieves a 75% compression ratio so it saves the transmission

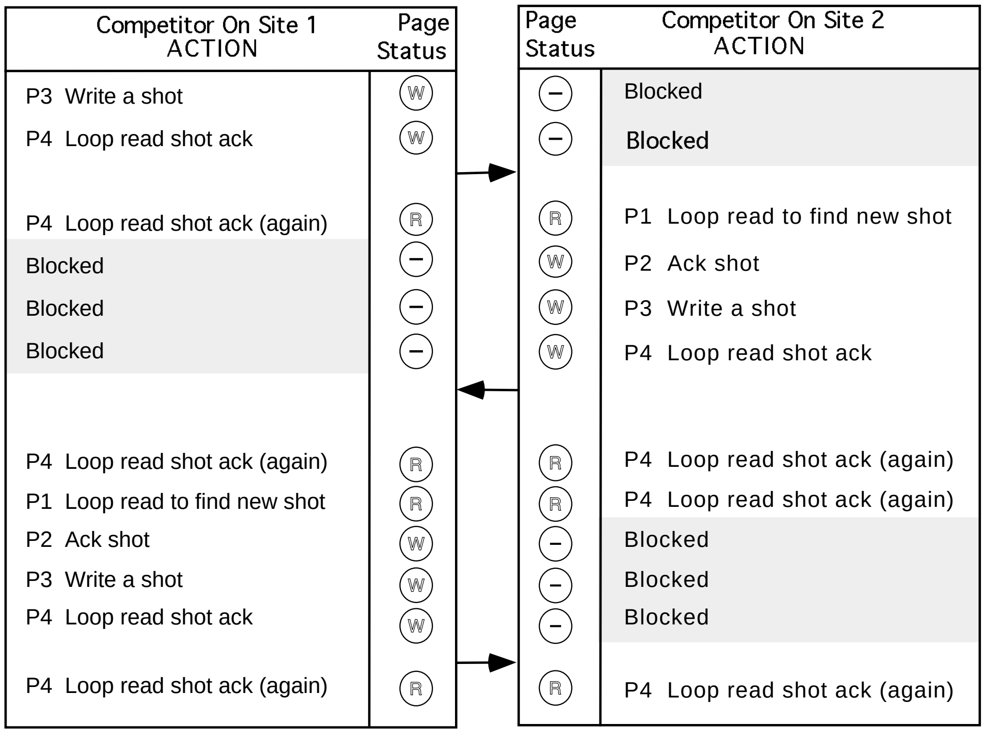
Competitor On Site 1 ACTION
Page Status
P3 Write a shot
P4 Loop read shot ack
P4 Loop read shot ack (again)
Blocked
Blocked
Blocked
P4 Loop read shot ack (again)
P1 Loop read to find new shot
P2 Ack shot
P3 Write a shot
P4 Loop read shot ack
P4 Loop read shot ack (again)
Page Status
Competitor On Site 2 ACTION
Blocked
Blocked
P1 Loop read to find new shot
P2 Ack shot
P3 Write a shot
P4 Loop read shot ack
P4 Loop read shot ack (again)
P4 Loop read shot ack (again)
Blocked
Blocked
Blocked
P4 Loop read shot ack (again)
W
R
−

Figure 5: Shared Memory Access Pattern of the Battleship Simulation

of one network message.

UptoN shows that the RLE algorithm works very well at compressing data in shared memory when the shared object does not consume the entire 4K page. In fact, for objects less than 3K in length the RLE algorithm will always achieve a 75% compression ratio.

RandN produces results which are almost twice as good as EachN. RandN begins sending uncompressed pages after about 650 random writes. Assuming a uniform distribution from the random number generator, this corresponds to one write every 6.3 bytes. In general, this experiment demonstrates that an application can write to a zeroed page about 600-700 times and still achieve some compression.

5.4 Performance of the Battleship Simulation

The purpose of this simulation was to provide a worst-case application with which we could exercise MIRAGE+ using two sites. The access pattern of our Battleship simulation follows the algorithm shown in Figure 3. We have implemented the simulation with one shared page for both competitors. The shared page holds two arrays representing battle fields. The observed access pattern for this type of sharing between the two competitors are shown in Figure 5.

Table 5 shows the performance based on whether the simulation runs on one or two sites, and whether competitors use the system call *yield* to voluntarily give up the CPU before the time slice is exhausted, thus permitting another process to execute. When running on one site, the yield call allows the competitor to fire a shot without waiting for the current time slice to expire. On one site, the observed behavior with respect to the yield call, as well as the time spent per iteration of the access pattern, are similar to the observations reported for Mirage[Fleisch 89b].

One site		Two different sites	
Without *yield*	Using *yield*	Implicit Δ locks	Explicit user locks
490 sec	4 sec	81 sec (51 msec)	48 sec (30 msec)

Table 5: Performance of the Battleship Simulation. Total Run-time.
Average time per turn per player in parenthesis.

The total run of the simulation contains 789 iterations, where each iteration has a shared memory access pattern as illustrated in Figure 5. Thus, the total run covers $2 * 789 = 1578$ executions of the following sequence: *a read fault*, *a page transfer*, *a write fault*, and *an invalidation*.

The work done by the simulation is dominated by the execution of this sequence. First, the read fault is sent to the library site where it is delayed Δ until the previous writer releases the page. Second, the page is transferred over the network, at the cost of approximately 20 msec including the request above (See Table 2). Third, the write fault occurs. It is also delayed Δ for an upgrade. In half of the instances, the faulting site is not the library site. Thus, a request is sent to the library site with cost comparable to an invalidation. The request is performed in parallel with the expiration of Δ. Finally, in all cases, an invalidation of one remote site occurs (4.6 msec). Therefore, using Δ=20 msec, the average time to execute the access pattern is $20 + 20 + 20 + 4.6 = 64.6$ msec. The measurement shown in Table 5 of 51 msec differs from the calculated time due to the implementation of *timesleep()*. In our implementation, Δ=20 msec is the smallest period a process can timesleep. However, the process may be awakened earlier, in which case, the operation ends. Had we required timesleep retry for a *minimum* of Δ, as in another experiment we performed, the measured execution time would be longer than the expected value.

We conducted several experiments with Δ=0 msec. Significant page stealing occurred between processes that had recently received pages, but not yet accessed them. In order to correct this problem, explicit checks were added to assure that a page was not relinquished before it has been accessed (for read faults) or written (for write faults). This improved performance, but added complexity to handle zero-filled page faults issued from remote sites when no memory has been allocated for the page (and thus accessed or written) anywhere in the cluster.

With Δ=0 msec, and by adding explicit MIRAGE+ write-lock system calls around the region where successive writes occur in the program, delays were eliminated due to under-utilization of the page during the Δ period. The average time it should take to execute the access pattern is $20 + \frac{1}{2} * 4.6 + 4.6 = 26.9$ msec. The measured time for two sites to execute one iteration of the access pattern in Figure 5 averaged 30 msec.

The relative performance figures of the battleship simulation running with and without compression are shown in Figure 6. Note that although the number of RLE messages transmitted is 54% less than the non-compressed number, the overall performance is enhanced 21%. The difference is attributed to the codec latency. Compression and decompression improves throughput by reducing the number of transmitted packets but increases the per-packet network latency.

5.5 Performance of Matrix Multiply

On our system, the matrix multiply algorithm achieved a near-linear speedup until communication costs became the significant factor. With compression enabled, we were able to achieve

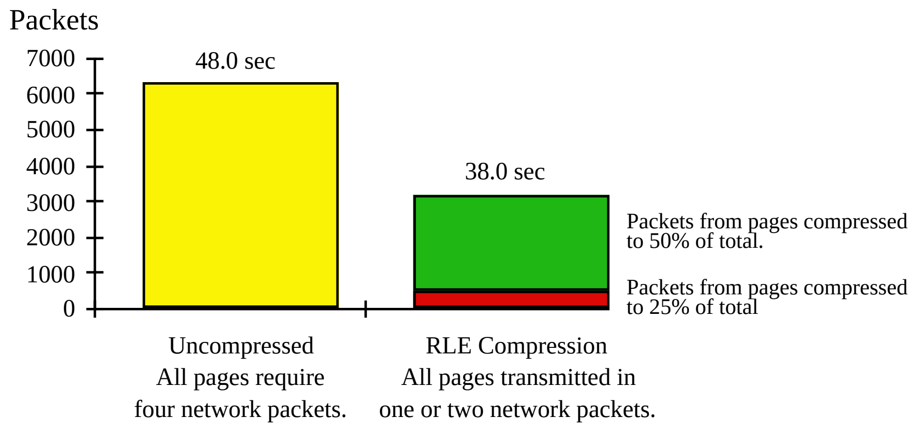

Figure 6: The Effect of Compression on Battleship Simulation
RLE Compression reduces the number of packets by 54%, and the total run-time by 21%

Number of sites	2	3	4	5	6	7
Matrix: 60×60	2.4	1.3	1.3	2.4	2.6	3.7
(with compression)	1.9	1.3	0.8	0.7	0.7	0.7
Matrix: 120×120	13.9	7.3	5.2	5.8	5.3	5.8
(with compression)	13.4	7.0	5.8	3.7	3.5	2.7
Matrix: 180×180	46.9	24.2	16.7	14.2	13.0	12.7
(with compression)	46.1	23.5	15.9	12.2	10.1	8.5
Matrix: 240×240	108.8	55.6	37.7	30.3	25.8	23.4
(with compression)	107.7	54.8	36.7	27.9	22.6	19.4
Matrix: 300×300	221.6	113.4	77.3	60.4	50.5	43.4
(with compression)	218.2	111.3	74.7	53.4	46.1	38.8
Matrix: 360×360	378.9	193.3	131.4	101.4	82.7	70.0
(with compression)	376.4	191.3	128.3	97.1	78.3	66.1
Matrix: 420×420	596.2	305.9	206.9	157.8	128.0	108.8
(with compression)	596.2	302.9	202.9	152.8	123.3	103.4
Matrix: 480×480	892.2	453.7	305.1	232.1	182.8	159.1
(with compression)	889.8	451.9	301.5	226.8	182.8	152.6

Table 6: Performance of Matrix Multiply
Measured in seconds, Δ=0 msec, packet blasting enabled, server proc handler.

a linear speedup with one or two additional sites working on each matrix size, assuming all pages compress to one packet. Table 6 provides the run times for the various experiments we ran on different matrix sizes. The speed-up of matrix multiply for the smallest and largest experiments we ran appears in Figure 7.

Figure 7 plots the speed-up obtained when using two through seven sites (one site which collects the data and one through six worker sites). It shows that in communication intensive applications (i.e., the 60×60 experiments), compression may improve performance dramatically. In compute intensive applications, the overall cost of using DSM communication is a

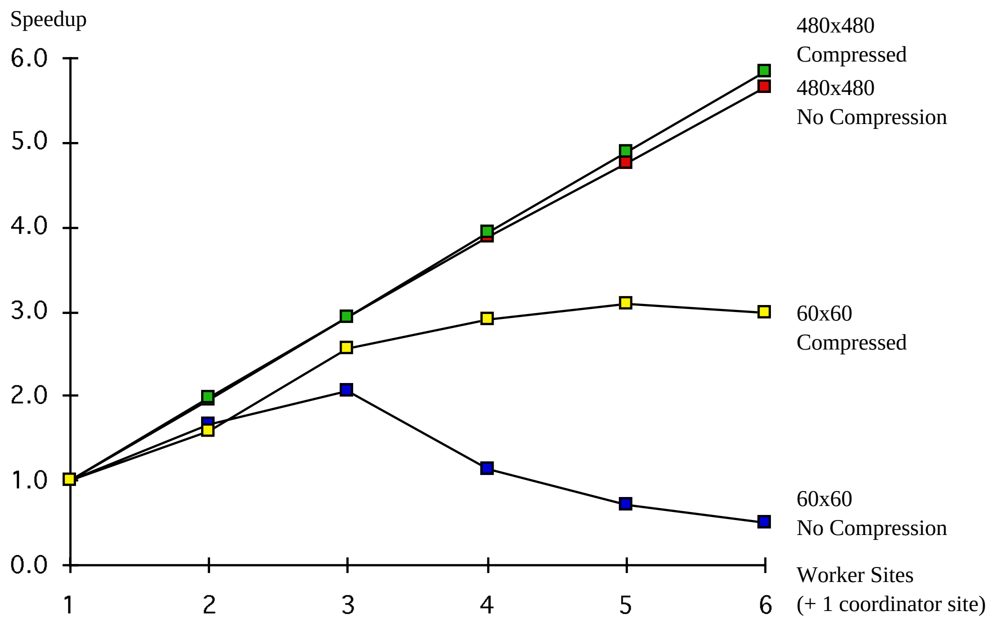

Figure 7: The Speed-up of Matrix Multiply

very small component of the overall cost, in agreement with our observations.

5.6 Performance Implications

The main lesson learned from our experiments is that the page size has a significant impact on overall performance for DSM systems. We optimized our use of the supplied communication subsystem by packet blasting and interrupt-level execution of packet reception without changing the communication subsystem. This reduced the average delay of a remote page fault from 31.7 msec to 16.7 msec. With compression added, the delay due to a remote page fault may be as short as 10.5 msec. Although our use of the communication subsystem has been tuned, we expect that enhancements to the network communication subsystem should improve performance further. This paper does not address these enhancements since our DSM system is designed to be a portable system, implemented on top of the basic network communication subsystem.

6 Related Work

In this section we describe work related to Mirage+. We focus on operating system implementations of DSM and do not discuss related hardware DSM systems and language/compiler implementations of DSM.

6.1 Ivy

Li[Li 86a, Li 88] experimented with a coherent shared virtual memory system on a loosely coupled multiprocessor, the Apollo Domain system[Nelson 84]. Shared data is paged between processors, some of which have copies of the virtual address space pages. The model assumes a write-invalidate DSM system where ownership of pages can vary from processor to processor either statically or dynamically. This work concentrates on consistency problems and theoretical performance based on experimentation with centralized and distributed managers to locate the page owner.

In contrast, our work is a kernel implementation in a commercial UNIX system. The model is based on paged segmentation. It gives the user additional flexibility in memory access and determining where to place data in memory. A time window mechanism is employed to avoid possible thrashing, to facilitate performance tuning, and to provide a locking mechanism. Our performance optimizations, as described in this paper, are not pursued in IVY.

6.2 Munin

Munin is a distributed shared memory system developed at Rice University[Bennett 89, Carter 93]. Munin is distinct in that it uses type-specific coherence mechanisms. These mechanisms are part of the run-time system and permit annotations from the user to specify the coherence mechanism to be used for each object. The authors based their design decisions on the results of a study of sharing and synchronization behavior in a variety of shared memory parallel programs[Bennett 90a].

The approach used in Munin is the antithesis of what we are doing in MIRAGE+. The MIRAGE+ experiment is to provide transparency to the System V IPC application designer concerning DSM application design. Munin makes the programmer aware of these aspects and requires that the programmer make designations of sharing characteristics before shared memory applications can expect to execute well. Munin's preliminary experiments with type-specific coherence protocols show that significant performance gains can be realized but that they are dependent on the characteristics of the parallel application. For example, the Matrix Multiply shows only a 4% improvement with annotated objects. However, other applications realized better than 50% execution time improvement.

6.3 Mether

Mether[Minnich 89] is a DSM system that supports inconsistent memory, leaving the responsibility for enforcing consistency to user-defined protocols. Mether operates on a cluster of SUN SPARCStations connected with Ethernet and running SunOS 4.0. DSM under Mether is implemented through modifications to the NSF file system. Mether maintains one consistent copy of a given page along with multiple inconsistent copies.

Although it is difficult to compare the systems directly, it is possible that the optimization techniques described in this paper could make MIRAGE+ competitive with systems that relax coherence and do not use our techniques.

7 Conclusions

This paper has addressed many issues in the design and implementation of a DSM system in our new environment; the paper relies heavily on our actual experiences with DSM. We have shown the complexity of the problems in moving memory management between environments and presented solutions that have work well for MIRAGE+.

7.1 Results

Our initial implementation of MIRAGE+ used synchronous communication for each of the four packets of a page. With a communication overhead of six msec per packet exchanged, the

System	Remote Page Fault (msec)	Battleship Simulation[1]	Matrix Multiply[2]
Mirage (S)	27.5	n/a	n/a
MIRAGE+ (S+H)	31.7	n/a	n/a
MIRAGE+ (S+B)	19.9	48 sec	2.4 sec
MIRAGE+ (I+B)	16.7	42 sec	2.0 sec
MIRAGE+ (S+B+C)	10.5/13.5/17.1/21.3	36 sec	1.9 sec
MIRAGE+ (I+B+C)	10.5/13.5/16.5/18.5	35 sec	1.8 sec

Table 7: Performance Summary

[1] Applications are run using explicit user locking and Δ=0. [2] 60×60 matrix running on two sites.
(H)andshake, (B)lasting packets, (I)nterrupt-level/(S)erver-procs, (C)ompression.

total time to fault a page across the network was 31.7 msec. By applying packet blasting, the page fault time was reduced 37% to 19.9 msec. Thus, we reduced the cost of sending the last three packets significantly. A further enhancement of running the reception of the first three packets as part of the interrupt handler reduces the cost by 16% to 16.6 msec.

For the applications and tests we have examined, the expected reduced performance in MIRAGE+ because of the larger page size and potential for false sharing is more than compensated by the faster CPU speeds. The larger page size reduces the number of pages transferred when there is no false sharing. Nonetheless, additional network messages must be sent to transfer the larger page. The challenging task for the DSM designer is to reduce the number of network transmissions while accommodating systems that have large page sizes.

With simple compression we have shown up to a 47.2% speed-up and a up to 75% reduction in the amount of network traffic. For our worst case application, the battleship simulation the speed-up was 21% and the packet reduction for page transfers were 54%. Depending on the compression ratio, the page fault time has been reduced to 10.5 msec for a page that compress into one packet, 13.5 msec for a page that compress into two packets, and 17.1 msec for a page that compress into three packets. A page that cannot be compressed costs 21.3 msec. Although we pay a penalty for pages that do not compress into less than three packets, compression has paid off substantially. Table 7 summarize our results.

7.2 Future Work

We plan to exercise additional MIRAGE+ applications and report the results. The MIRAGE+ compression succeeded insofar as it demonstrated the feasibility of compression to improve the performance of a DSM system. We assume the run length encoding algorithm in MIRAGE+ compression is not the best choice. We plan to examine different compression algorithms with a representative set of applications and compare the various performance benefits of each compression technique. Work is currently underway to develop special compression algorithms which complement MIRAGE+ memory access patterns and high speed networks.

MIRAGE+ reliability is another issue that we are currently addressing. Our goal is to have a working version of MIRAGE+ that is tolerant of single site failures using techniques that could enhance performance and improve reliability.

8 Acknowledgements

The fine-grain profiling of the kernel was made possible by a micro-timing function created by of Noah Mendelsohn of IBM. Cheryl DeMatteis and Carson Ellsworth developed additional applications to exercise Mirage+.

Thanks to the anonymous referees for asking us to clarify several explanations and streamline the presentation. During the revision process we improved overall performance by a factor of two for some code in the system. Thanks to Cheryl DeMatteis, who also reviewed the paper.

This research was sponsored by a Joint Study with IBM Corporation and is currently sponsored by NSF CCR-9209405. Juul was also supported by the Danish Natural Science Research Council.

A preliminary version of this paper, with early results, has been released as UC, Riverside Technical Report UCR-CS-93-6[Fleisch 93].

References

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, USENIX, June 1986.

[Bell 89a] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, USA, 1989.

[Bell 89b] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–589, December 1989.

[Bennett 89] John K. Bennett, John B. Carter, and Willy Zwaenepoel. *Munin: Shared Memory for Distributed Memory Multiprocessors*. Technical Report COMP TR89-91, Rice University, April 1989.

[Bennett 90a] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[Bennett 90b] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 Conf. Principles and Practice of Parallel Programming*, pages 168–176, ACM Press, New York, NY, USA, 1990.

[Bisiani 90] Roberto Bisiani and Mosur Ravishankar. *PLUS: A Distributed Shared-Memory System*. Technical Report, School of Computer Science, Carnegie Mellon University, 1990.

[Brinch Hansen 73] Per Brinch Hansen. *Operating Systems Principles*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, July 1973.

[Carter 89] John B. Carter and Willy Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In International Conference on Measurement and Modeling of Computer Systems, Proceedings in: *Performance Evaluation Review* Volume 17(1), pages 61–6, Berkeley, CA, USA, May 1989.

[Carter 93] John B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, Houston, Texas, USA, September 1993.

[Daley 68] R. C. Daley and J. B. Dennis. Virtual memory processes and sharing in Multics. *Communications of the ACM*, 11(5):306–311, May 1968.

[Denning 70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[Dijkstra 72] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes.* Academic Press, New York, NY, USA, 1972.

[Fleisch 89a] Brett D. Fleisch. *Distributed Shared Memory in a Loosely Coupled Environment.* PhD thesis, Computer Science Department, University of California, Los Angeles, CA, USA, September 1989.

[Fleisch 89b] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, published in *Operating Systems Review* 23(5) Special Issue, pages 211–223, ACM SIGOPS, ACM Press, The Wigwam, Litchfield Park, Arizona, USA, December 1989.

[Fleisch 93] Brett D. Fleisch, Randall L. Hyde, and Niels Christian Juul. *Moving Distributed Shared Memory to the Personal Computer: The* MIRAGE+ *Experience.* Technical Report UCR-CS-93-6, Department of Computer Science, University of California, Riverside, CA, USA, June 1993.

[Hoare 74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):545–57, October 1974.

[Hyde 94] Randall L. Hyde and Brett D. Fleisch. *An Analysis of Degenerate Sharing and False Coherence.* Technical Report UCR-CS-94-1, Department of Computer Science, University of California, Riverside, CA, USA, January 1994.

[Li 86a] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[Li 86b] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings 5th ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pages 229–239, ACM Press, Canada, August 1986.

[Li 88] Kai Li. IVY: a shared virtual memory system for parallel computing. In *Proceedings 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[Li 90] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *IEEE Computer*, 23(5):54–64, May 1990.

[Metcalfe 76] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–403, 1976.

[Minnich 89] Ronald G. Minnich and David J. Farbar. The Mether system: Distributed shared memory for SunOS 4.0. In *Proceedings of the Summer 1989 USENIX Conference*, pages 51–60, USENIX, Baltimore, Maryland, USA, June 1989.

[Nelson 84] David L. Nelson and Paul J. Leach. The architecture and applications of the Apollo Domain. *IEEE Computer Graphics and Applications*, 58–66, April 1984.

[Nelson 91] M. Nelson. *The Data Compression Book.* M & T Books, Redwood City, CA, USA, 1991.

[Nitzberg 91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A network transparent, high reliability distributed system. In Proceedings of the Eigth ACM Symposium on Operating Systems Principles, published in *Operating Systems Review* 15, pages 169–177, ACM SIGOPS, ACM Press, Pacific Grove, CA, USA, December 1981.

[Raita 87] T. Raita and J. Teuhola. Predictive text compression by hashing. In *Proceedings of the 10th Annual ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 223–233, ACM, New Orleans, USA, June 1987.

[Ramachandran 88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. *Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory.* Technical Report GIT-ICS-88/23, Georgia Institute of Technology, Atlanta, GA, USA, June 1988.

[Schroeder 90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Storer 88] J. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, Maryland, USA, 1988.

[Tanenbaum 92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992.

[Thekkath 93] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[Walker 83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In Proceedings of the Nineth ACM Symposium on Operating Systems Principles, published in *Operating Systems Review* 17(5), pages 49–70, ACM SIGOPS, ACM Press, Bretton Woods, NH, USA, October 1983.

[Williams 90] R. Williams. *Adaptive Data Compression.* Kluwer Books, Norwell, Ma, USA, 1990.

[Ziv 77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[Zwaenepoel 85] Willy Zwaenepoel. *Protocols for Large Data Transfers over Local Networks.* Technical Report COMP TR85-23, Department of Computer Science, Rice University, Houston, Texas, USA, July 1985.